Not All Hackers Are Evil: Automated ReDos Generator for Detecting Vulnerable Regexes

LENKA TUROŇOVÁ, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
MARGUS VEANES, Microsoft, USA
TOMÁŠ VOJNAR, Brno University of Technology, Czech Republic

Regular expression matching can have exponential runtime and may lead to an algorithmic complexity attack known as denial-of-service (ReDoS) attack. One way, how the attacker may launch ReDoS attack is to carefully craft an input string that matches the theoretical worst-case performance of the matching algorithm. This paper presents a tool `Cavil` which automatically generates evil strings to detect vulnerabilities of given regexes. Our technique especially focuses on regular expressions (regexes) with bounded repetition, such as $(ab)\{100\}$. `Cavil` constructs a deterministic counting-set automaton (*CsA*) and then uses guided search techniques that exploits the state space towards the maximal counting loops. We evaluated it against 5540 regexes from real-world SW projects. The evaluation results show that `Cavil` found 4 times more attack strings compared to the best state-of-the-art techniques.

CCS Concepts: • **Theory of computation → Regular languages**; *Quantitative automata*; • **Security and privacy → Denial-of-service attacks**; • **Applied computing** → *Document searching*.

Additional Key Words and Phrases: regular epxression matching, bounded repetition, ReDos, determinization, Antimirov's derivatives, counting automata, counting-set automata

ACM Reference Format:
Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2020. Not All Hackers Are Evil: Automated ReDos Generator for Detecting Vulnerable Regexes . *Proc. ACM Program. Lang.* 4, OOPSLA, Article 218 (November 2020), **??** pages. https://doi.org/10.1145/3428286

1 INTRODUCTION

Matching *regexes* (regular expressions) is a ubiquitous component of software, used, e.g., for searching, data validation, parsing, finding and replacing, data scraping, or syntax highlighting. It is commonly used and natively supported in most programming languages [**?**]. For instance, about 30–40 % of Java, JavaScript, and Python software use regex matching (as reported in multiple studies, see, e.g., [**?**]).

The efficiency of regex matching engines has a significant impact on the overall usability of software applications. Unpredictability of a matcher's performance may lead to catastrophic

Authors' addresses: Lenka Turoňová, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, ituronova@fit.vutbr.cz; Lukáš Holík, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, holik@fit.vutbr.cz; Ondřej Lengál, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, lengal@fit.vutbr.cz; Margus Veanes, MSR, Microsoft, One Microsoft Way, Redmond, 98052, USA, margus@microsoft.com; Tomáš Vojnar, Faculty of Information Technology, Brno University of Technology, Božetěchova 2, Brno, 612 00, Czech Republic, vojnar@fit.vutbr.cz.

2475-1421/2020/11-ART218
https://doi.org/10.1145/3428286

consequences, witnessed by events such as the recent catastrophic outage of Cloudflare services [?], caused by a single poorly written regex, and it is a doorway for the so-called ReDoS attack, a denial of service attack based on overwhelming a regex matching engine by providing a specially crafted regex or text. For instance, in 2016, ReDoS caused an outage of StackOverflow [?] or rendered vulnerable websites that used the popular Express.js framework [?]. Works such as [??] give arguments that ReDoS is not just a niche problem but rather a common and serious threat.

Failures of matching are mostly caused by the so-called "catastrophic backtracking", a situation when variants of Spencer's simulation of a *nondeterministic finite automaton* (NFA) by *backtracking* [?] exhibit a behaviour super-linear to the length of the text. Matching algorithms based on backtracking are probably the most often implemented ones, their performance is, however, at worst *exponential* to the text length. An alternative with a much lower worst-case complexity (wrt the length of the text) is to use *deterministic finite automata* (DFAs). In the ideal case, the DFA is pre-computed; matching can then be linear to the text length, with each input symbol processed in constant time. This is the so-called *static DFA simulation* [?]. The major drawback of static DFA simulation is that the DFA construction may explode, rendering the method unusable in practice.

Variants of Thompson's algorithm [?] (sometimes called NFA simulation or NFA-to-DFA simulation) avoid the explosion by working directly with the NFA. They essentially run the determinization by subset construction *on the fly*, always remembering only the current DFA state. On reading a character, a successor DFA state is computed and used to replace the current state. The disadvantage of Thompson's algorithm is that, for a highly nondeterministic NFA, the DFA states—sets of the states of the NFA—may get large and computing a DFA-state successor over a symbol becomes expensive, linear to the size of the NFA (compared to the constant time of static DFA simulation).

Modern matchers therefore use caching of already visited parts of the DFA. Making a step within the cached part is then as fast as with the explicitly determinized automaton. Extremely efficient implementations of Thompson's algorithm with caching are used in `RE2` [?] and GNU `grep` [?]. Their close cousin, an on-the-fly Brzozowski's derivative construction, is implemented in the tool `SRM` [?]. Highly nondeterministic regexes[1] that lead to exploding determinization are, however, problematic for all variants, explicit determinization as well as NFA simulation, with or without caching.

In this paper, we focus on regulare expressions with the *counting operator*, also known as the operator of *bounded repetition*. It succinctly expresses repeated patterns such as `(ab){1,100}`, representing 1 to 100 consecutive repetitions of `ab`. Such expressions are very common (cf. [?]), e.g., in the RegExLib library [?], which collects expressions for recognizing URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [?] used for detecting attacks in network traffic; in real-life XML schemas, with the counter bounds being as large as 10 million [?]; or in detecting information leakage from traffic logs [?]. These regular expression are more likely to be vulnerable again ReDoS attacks.

To illustrate the principal difficulty with matching bounded repetitions, especially when combined with a high degree of nondeterminism, consider the regex `.*a.{`k`}` where $k \in \mathbb{N}$ (the regex denotes strings where the symbol `a` appears k positions from the end of the word). Already the NFA will have at least k states, which is exponential to the regex size because k is written in decimal. Due to the inherent nondeterminism of this regex, determinization then adds a second level of the exponential explosion. Indeed, the minimal DFA accepting the language has 2^{k+1} states because it must remember all the positions where the symbol `a` was seen during the last $k + 1$ steps. This

[1] Loosely speaking, a "highly nondeterministic regex" is one for which the determinization of the NFA created by some of the usual algorithms explodes. Determinism of regexes closely corresponds to the notion of *1-unambiguity of the regex* [??]: when matching a text from left to right against the regex, it is always clear which letter of the regex matches the text character.

requires a finite memory of $k + 1$ bits and thus 2^{k+1} reachable DFA states. Determinizing the NFA explicitly is thus out of question for even moderate values of k. The pure Thompson's NFA simulation is feasible but very slow, as reading each character may in the worst case require processing the entire NFA. Moreover, caching of the DFA state space, used in industrial matchers like `RE2` [?] or GNU `grep` [?], may also be ineffective due to the size of the state space and low cache utilization. At the same time, combinations of nondeterminism and counting are fairly common. A high degree of nondeterminism is, for instance, usual when searching for a pattern "anywhere on the line" (corresponding to prefixing the pattern with `.*`), which is the standard behaviour for GNU `grep` and similar programs when start/end of line anchors are not used.

2 PRELIMINARIES

We cast our definitions in the framework of symbolic automata [?], a natural succinct representations of finite-state transition relations over large alphabets of labels. Symbolic automata work over alphabets equipped with a so-called effective Boolean algebra, which defines the needed interface for handling large sets of labels on automata transitions.

Before providing the formal definition of an effective Boolean algebra, we start with an intuitive example, which is also going to be the alphabet algebra used throughout the paper, including the experiments. Later on, we will further leverage the general definition to work with algebras over counter and counting-set predicates.

Example 2.1. Regular expressions in practice use *character classes* as basic building blocks. To simplify the discussion, let us restrict our attention to ASCII as the character universe $\mathfrak{D}$. In other words, $\mathfrak{D}$ is the set $\{n \mid 0 \leq n < 2^7\}$ of all 7-bit characters represented using their character codes. Then, for example, the character classes `[0-9]` and `[A-Z]` denote, respectively, the set $[\![\texttt{[0-9]}]\!] = \{n \mid 48 \leq n \leq 57\}$ of all digit codes, and the set $[\![\texttt{[A-Z]}]\!] = \{n \mid 65 \leq n \leq 90\}$ of all upper-case letter codes. Character classes made up of individual symbols such as `@` denote singleton sets, e.g., $[\![\texttt{@}]\!] = \{64\}$. Character classes can also be used to form *unions*, they can be *complemented*, and even *subtracted* from each other, and are in general closed under Boolean operations. There are therefore many different ways how to represent the same character sets, e.g., $[\![\texttt{[0-9]}]\!] = [\![\texttt{[0-45-9]}]\!] = [\![\texttt{[0-4]}]\!] \cup [\![\texttt{[5-9]}]\!]$. To illustrate the complement, for example, `[^0-9]` denotes the set of all non-digits, as does `[\x00-\x2F\x3A-\x7F]`. The set of all character classes is then an example of the set Ψ of all *predicates* of a Boolean algebra, and checking *satisfiability* of a predicate $\varphi \in \Psi$ means to decide whether φ denotes a *nonempty* set. For example, the predicate `[]` is unsatisfiable because $[\![\texttt{[]}]\!] = \emptyset$ and `.` denotes the *true* predicate because $[\![\texttt{.}]\!] = \mathfrak{D}$. Further, note that a character class can, without loss of generality, be represented as a Boolean combination of *intervals* or even as a union of intervals if normalized. □

2.1 Effective Boolean Algebras

An *effective Boolean algebra* $\mathbb{A}$ has components $(\mathfrak{D}, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where $\mathfrak{D}$ is a *universe* of underlying domain elements. Ψ is a set of unary *predicates* closed under the Boolean connectives $\vee, \wedge : \Psi \times \Psi \to \Psi$ and $\neg : \Psi \to \Psi$; and $\bot, \top \in \Psi$ are the *false* and *true* predicates. Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ satisfies that $[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathfrak{D}$, and for all $\varphi, \psi \in \Psi$, $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$, $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and $[\![\neg\varphi]\!] = \mathfrak{D} \setminus [\![\varphi]\!]$. For $\varphi \in \Psi$, we write $\textbf{\textit{Sat}}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*. We require that ***Sat*** as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g., $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\mathfrak{D}_{\mathbb{A}}}$.

2.2 Words and Regexes

The basic building blocks of regexes are *predicates* from an effective Boolean algebra *CharClass* of *character classes*, such as the class of digits, written as \d. Let $\mathfrak{D} = \mathfrak{D}_{CharClass}$. A *word* over $\mathfrak{D}$ is a sequence of symbols $a_1 \cdots a_n \in \mathfrak{D}^*$ and a *language* $\mathcal{L}$ over $\mathfrak{D}$ is a subset of $\mathfrak{D}^*$. We use ϵ to denote the *empty word*. The concatenation of words u and v is denoted as $u \cdot v$ (often abbreviated to uv) and is lifted to sets as usual. We call $a \in \mathfrak{D}$ the *head* of the word $a.w$ and $w \in \mathfrak{D}^*$ its *tail*. Furthermore, we write $\mathcal{L}^n$ for the n-th power of $\mathcal{L} \subseteq \mathfrak{D}^*$ with $\mathcal{L}^0 \stackrel{\text{def}}{=} \{\epsilon\}$ and $\mathcal{L}^{n+1} \stackrel{\text{def}}{=} \mathcal{L}^n \cdot \mathcal{L}$.

The abstract syntax of regexes is the following with $\alpha \in \Psi_{CharClass}$ and n, m being integers such that $0 \leq n$, $0 < m$, and $n \leq m$:

$$\varepsilon \quad \alpha \quad R_1 \cdot R_2 \quad R_1|R_2 \quad R\{n,m\} \quad R*$$

where $R_1 \cdot R_2$ is called a *concat node* and $R_1|R_2$ is called a *choice node*. The semantics of a regex R is defined as a subset of $\mathfrak{D}^*$ in the following way: $\mathcal{L}(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$, $\mathcal{L}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\}$, $\mathcal{L}(R_1R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, $\mathcal{L}(R_1|R_2) \stackrel{\text{def}}{=} \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} (\mathcal{L}(R))^i$, and $\mathcal{L}(R*) \stackrel{\text{def}}{=} \mathcal{L}(R)^*$. R is *nullable* if $\epsilon \in \mathcal{L}(R)$. We will also need to refer to the number of *character-class leaf nodes* of a regex R, denoted by $\sharp_\Psi(R)$ and defined as follows: $\sharp_\Psi(\varepsilon) = 0$, $\sharp_\Psi(\alpha) = 1$, $\sharp_\Psi(R_1 \cdot R_2) = \sharp_\Psi(R_1|R_2) = \sharp_\Psi(R_1) + \sharp_\Psi(R_2)$, $\sharp_\Psi(R\{n,m\}) = \sharp_\Psi(R*) = \sharp_\Psi(R)$.

2.3 Minterms

Let $Preds(R)$ be the set of all predicates that occur in a regex R, and let $Minterms(R)$ denote the set of *minterms* of $Preds(R)$. Intuitively, $Minterms(R)$ is a set of non-overlapping predicates that can be treated as a concrete finite alphabet. Each minterm is essentially a region in the Venn diagram of the predicates in R: it is a satisfiable conjunction $\bigwedge_{\psi \in Preds(R)} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. For example, if $R =$ `[0-z]{4}[0-8]{5}`, then $Preds(R) = \{$`[0-8]`, `[0-z]`$\}$ and $Minterms(R) = \{$`[0-8]`, `[9-z]`, `[^0-z]`$\}$. Formally, if $\alpha \in Minterms(R)$, then $\boldsymbol{Sat}(\alpha)$ and $\forall \psi \in Preds(R)$: $[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!]$.

Note that although the number of minterms of a general set X of predicates may be exponential in $|X|$, it is only linear if X consists of intervals of symbols used in regexes, such as `[a-zA-Z]` or `[^a-zA-Z]` (the former denotes two intervals while the latter their complement, which is equivalent to the union of three intervals). Intervals of numbers generate only a linear number of minterms.

2.4 Symbolic Automata

We use *symbolic finite automata (FAs)*, whose alphabet is given by an effective Boolean algebra, as a generalization of classical finite automata. Formally, an FA is a tuple $A = (\mathbb{I}, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the input algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ is a finite set of transitions. A transition $(q, \alpha, r) \in \Delta$ will be also written as $q\text{-}(\alpha)\!\rightarrow\! r$.

A *run of A from a state p_0* over a word $a_1 \cdots a_n$ is a sequence of transitions $(p_{i-1}\text{-}(\alpha_i)\!\rightarrow\! p_i)_{i=1}^n$ with $a_i \in [\![\alpha_i]\!]$; the run is *accepting* if $p_n \in F$. The *language of a state q*, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q. The *language of A*, denoted $\mathcal{L}(A)$, is $\mathcal{L}_A(q_0)$. A classical finite automaton can be understood as an FA where the basic predicates have singleton set semantics, i.e., when for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$. A is *deterministic* iff for all $p \in Q$ and all transitions $p\text{-}(\alpha)\!\rightarrow\! q$ and $p\text{-}(\alpha')\!\rightarrow\! r$, it holds that if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$.

3 COUNTING AUTOMATA

Counting automata (CAs) are a natural and compact automata counterpart for regexes with counting. They are essentially a limited sub-class of classical counter automata, in which counters are only

supposed to count the number of passes through some of its parts (corresponding to a counted sub-expression of a regex) and guards on transitions enforce a specified number of repetitions of that part before the automaton is allowed to move on.

Counter algebra. A *counter algebra* is an effective Boolean algebra $\mathbb{C}$ associated with a finite set C of counters. The counters play the role of bounded loop variables associated with a *lower bound* $\boldsymbol{min}_c \geq 0$ and an *upper bound* $\boldsymbol{max}_c > 0$ such that $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. $\mathfrak{D}_\mathbb{C}$ is the set of interpretations $\mathfrak{m} : C \to \mathbb{N}$, called *counter memories* such that $0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$ for all $c \in C$. $\Psi_\mathbb{C}$ contains Boolean combinations of *basic* predicates $\textsc{canExit}_c$ and $\textsc{canIncr}_c$, for $c \in C$, whose semantics is given by

$$\mathfrak{m} \models \textsc{canExit}_c \iff \mathfrak{m}(c) \geq \boldsymbol{min}_c, \quad \mathfrak{m} \models \textsc{canIncr}_c \iff \mathfrak{m}(c) < \boldsymbol{max}_c.$$

Counting automata. A *counting automaton (CA)* is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*, C is a finite set of *counters* with an associated counter algebra $\mathbb{C}$, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $F : Q \to \Psi_\mathbb{C}$ is the *final-state condition*, and $\Delta \subseteq Q \times \Psi_\mathbb{I} \times (C \to \boldsymbol{O}) \times Q$ is the (finite) *transition relation*, where $\boldsymbol{O} = \{\textsc{exit}, \textsc{incr}, \textsc{exit1}, \textsc{noop}\}$ is the set of *counter operations*. The component f of a transition $p\text{-}(\alpha,f)\!\!\rightarrow\! q \in \Delta$ is its *(counter) operator*. We often view f as a set of *indexed operations* $\textsc{op}_c$ to denote the operation assigned to the counter c, $f(c) = \textsc{op}$.

Semantics of CAs. The semantics of the CA A is defined through its *configuration automaton* $FA(A)$, an FA whose states are A's *configurations*, i.e., pairs $(q, \mathfrak{m}) \in Q \times \mathfrak{D}_\mathbb{C}$ consisting of a state q and a counter memory $\mathfrak{m}$. To define $FA(A)$, we must first define the semantics of counter operators f, which occur on transitions. For this, we associate with each (indexed) operation $\textsc{op}_c$ a counter guard $grd(\textsc{op}_c)$ and a counter update $upd(\textsc{op})$ as shown on the right. Intuitively, the operation NOOP does not modify the counter's value and is always enabled. The operation INCR increments the counter and is enabled if the counter has not yet reached its upper bound. The operation EXIT resets the counter to 0 on exit from the counting loop and is enabled when the counter reaches its lower bound. The operation EXIT1 executes EXIT followed by INCR. The *guard* of a counter operator $f : C \to \boldsymbol{O}$ is then a predicate $\varphi_f \in \Psi_\mathbb{C}$ over counter memories, and its *update* $\mathrm{f} : \mathfrak{D}_\mathbb{C} \cup \{\bot\} \to \mathfrak{D}_\mathbb{C} \cup \{\bot\}$ is a counter-memory transformer:

$$\begin{array}{llll} grd(\textsc{noop}_c) \stackrel{\text{def}}{=} \top_\mathbb{C} & & upd(\textsc{noop}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n \\ grd(\textsc{incr}_c) \stackrel{\text{def}}{=} \textsc{canIncr}_c & & upd(\textsc{incr}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n+1 \\ grd(\textsc{exit}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c & & upd(\textsc{exit}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.0 \\ grd(\textsc{exit1}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c & & upd(\textsc{exit1}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.1 \end{array}$$

$$\varphi_f \stackrel{\text{def}}{=} \bigwedge_{\textsc{op}_c \in f} grd(\textsc{op}_c) \qquad \mathrm{f}(\mathfrak{m}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{\lambda} c.upd(f(c))(\mathfrak{m}(c)) & \text{if } \mathfrak{m} \models \varphi_f \\ \bot & \text{otherwise} \end{cases}$$

Intuitively, f updates all counters in a counter memory $\mathfrak{m}$ by their corresponding operations if $\mathfrak{m}$ satisfies the guard, otherwise the result is $\bot$.

We now define the *configuration automaton* of A, denoted as $FA(A)$, which defines the language semantics of the CA A. The states of $FA(A)$ are the configurations of A (there are finitely many of them), and the initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto 0 \mid c \in C\})$ of A. A state $(p, \mathfrak{m})$ of $FA(A)$ is *final* iff $\mathfrak{m} \models F(p)$. The transition relation $\Delta_{FA(A)}$ of $FA(A)$ is defined as

$$\Delta_{FA(A)} = \{(p, \mathfrak{m}) \text{-}(\alpha)\!\!\rightarrow\! (q, \mathrm{f}(\mathfrak{m})) \mid p \text{-}(\alpha,f)\!\!\rightarrow\! q \in \Delta, \mathfrak{m} \models \varphi_f\}.$$

Deterministic and simple CAs. A is *deterministic* iff the following holds for every state $p \in Q$ and every two transitions $p\text{-}(\alpha_1,f_1)\!\!\rightarrow\! q_1$, $p\text{-}(\alpha_2,f_2)\!\!\rightarrow\! q_2 \in \Delta$: if both $\alpha_1 \wedge \alpha_2$ and $\varphi_{f_1} \wedge \varphi_{f_2}$ are satisfiable, then $f_1 = f_2$ and $q_1 = q_2$. It follows from the definitions that, if A is deterministic, then $FA(A)$ is deterministic too. A is *simple* if for any two transitions $q\text{-}(\alpha,f)\!\!\rightarrow\! r$ and $q'\text{-}(\alpha',f')\!\!\rightarrow\! r'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. That is, different character guards do not overlap and can be mostly treated as

plain symbols. We also require that all guards are satisfiable. CAs constructed from regexes by the algorithm in **??** will be simple.

4 REDOS ATTACKS

Regular expressions are widely used in many programming languages. Unfortunately, they are only small parts of a big projects so they often lack testing and analysing. However, a single poorly-written regex may lead to catastrophic consequences, such as failed input validation, leaky firewalls and even a ReDoS attack. Therefore, there is a desire to develop an automated tool for analyzing the regexes to protect systems against these attacks.

The stress testing is an active area of research. The most of the state-of-the-art ReDoS analyzers focus on regex matchers that are based on backtracking algorithm. The weakness point of the backtracking matchers is that they may have a running time that is exponential in the size of the input. The naive algorithm for matching regular expression is to build a nondeterministic finite automaton (NFA) such that there can be several possible next states for each pair of state and input symbol. The backtracking algorithm tries all the possible paths until it finds a match.

Example 4.1. To illustrate the vulnerability of the regex matchers based on backtracking algorithm, let's have a regular expression $\hat{}(a+)+\$$ and an input text $a^n b$. A backtracking matcher would take an exponential time in n when trying to find a match. All attempts fail in the end since there is a letter b at the end of the input. For $n = 4$ there are 16 possible runs. For $n = 16$ there are 65636 possible runs. This is an extreme case where the algorithm must pass many path to fail.

The state-of-the-art ReDoS generators use this knowledge to generate a well-crafted input that will make the matcher vulnerable. The generator can be either static, dynamic or combination of both based on whether actual regex matching is conducted.

Static ReDoS generators are based purely on the regex. They represent a given regex by e-NFA [**?**]. They use different techniques - e.g., pumping analysis [**???**], transducer analysis [**?**], adversarial automata construction [**?**], or NFA ambiguity analysis [**?**]. They can be sound and complete for certain class of regexes, however, the major disadvantage of these generators is that often report false positive attacks, or miss ReDoS vulnerability of regexes.

*The tool RegexStatic [**?**] is the first representative of the static generators. It creates pNFA to develop an NFAA tool for detecting ReDoS vulnerabilities for regexes. pNFA is a new automaton model, which can describe notions like back-references and named groups. However, new attack string patterns based on this model need exploration and improvement.*

The next tool based on the static analysis is Rexploiter [**?**]. Given a regular expression, it constructs a corresponding NFA based on which determines the worst-case complexity of matching the regex against an arbitrary input string. The algorithm can identify whether an NFA has linear, super-linear, or exponential time complexity. Moreover, unlike RegexStatic [**?**], it can also construct an attack automaton in order to capture all possible attack strings that trigger worst-case behaviour of the matching algorithm.

The main idea of the analysis is that a NFA A is hyper-vulnerable (has exponential complexity) if there exists a string s such that the number of distinct matching paths π_i (labelled with the same symbols) leading from state q_0 to a rejecting state q_r is exponential in the length of s. Hence, if A rejects s then the backtracking matching algorithm needs to explore each of these exponentially many paths.

The A has to contain a pivot state q reachable from the initial state q_0, such that there are two paths π_1 and π_2 back to the state q on the same input string s. Moreover, there has to be a way of reaching a rejecting state q_r from q. The result is a string $s = s_0 \cdot s_c^k \cdot s_1$ with a *attack prefix* s_0 given by $labels(\pi_p)$, an *attack suffix* s_1 given by $labels(\pi_s)$, and *attack core* s_c given by $labels(\pi_1)$,

that will be rejected by A. Clearly, with increasing value of k the running time of the backtracking matching algorithm will double.

Example 4.2. Let's have a regular expression $(a|aa)^*\$$ **??**. The corresponding NFA A is hyper-vulnerable since there exist paths $\pi_1 = (q, a, q), (q, a, q)$ and $\pi_2 = (q, a, q_0), (q_0, a, q)$ labelled with the same symbols. 1) Both runs start and end in the pivot state q, 2) q is reachable from q_0, and 3) there exists a way from q to rejecting state q_r.

Based on the analysis, an attack automaton $A^E = A_p \cdot (A_1 \cap A_2) \cdot \overline{A_s}$ which represents all attack strings for the given regular expression such that A_p accepts all prefixes of A which end up in q; A_s accepts all suffixes starting from q and end up in q_r, and A_1 (resp. A_2) corresponds to a set of paths that loop back to q.

Even though it is considered to be a static analyzer it also uses dynamic analysis to avoid generating false positives. During the additional analysis, it determines for each regular expression a lower bound on the length k of any possible attack string. In addition to the analysis, it also inducts sanitization-aware taint analysis at the source code level. Even, it could be successful in case of a certain group of regexes, it is not able to handle regexes with extended grammars (except from gready quantifiers $\{m, n\}$, $\{n\}$, or lazy quantifiers $??, +?$).

A tool RXXR2 [**??**] creates an e-NFA from a given regex and then it search for instances of a pattern in the e-NFA using an efficient pattern matching algorithm. It is written in OCaml. Similarly to Rexploiter, it is unable to parse extended regexes (except from backreferences, non-capturing groups, greedy quantifiers and lazy quantifiers).

While, dynamic ReDoS generators conduct actual regex matching and use the profiling results to improve next iteration of text generating, e.g., fuzzing [**?**]. Thus, they usually easily handles regex extensions and reports only true positive ReDoS input strings. However, the main drawback of the dynamic generators is that they are much more time and space consuming so they may miss ReDoS vulnerabilities in case of complex regexes due to the time or space limits.

To the best know tools using dynamic fuzzing belongs SlowFuzz [**?**]. The base of the tool is a evolutionary fuzzer [**?**] to search for those inputs that can trigger a large number of edges in the control flow graph of the program under testing. However, SlowFuzz lacks knowledge about regex structures, and this fact can cause some deep states of the program unreachable, leading to false negatives. It is written in C. The SlowFuzz is the most general tool for generating an even texts, since it can handle extended grammar (set operations $[a-z]\&\&[\hat{}aeiou]$, lookarounds $(?=), (?<=)$, backreferences $\backslash 1$, non-capturing groups $(?:)$, named groups, atomic groups $(?>)$, gready quantifiers $\{m, n\}$, $\{n\}$, $\{m,\}$, lazy quantifiers $??, *?, \{\}?$, conditionals $()?, (?(1))$, or possessive quantifiers $?*, *+, \{\}+$).

A tool SDLFuzzer is another dynamic fuzzer for ReDoS detection []. It was developed by Bryan Sullivan from Microsoft in 2010. However, this tool has no longer been maintained and has no further update since then. Therefore, the tool does not support many extended regex grammars, e.g., lookarounds, named groups, atomic groups, back-references, and lazy or possessive quantifiers.

In between these tools, there exist tools that combine dynamic and static techniques, e.g. Rescue [**?**]. Rescue (written in Java) uses the static information about the regex in a form of extended NFA (e-NFA) and a genetic search to guide the subsequent genetic search. Like Slowfuzz, Rescue can handle also extended grammar (except from set operations and conditionals).

The aim of this technique is to find an input string that maximizes the number of matching steps using regex search profiling data. While the classical matching technique keeps an e-NFA state and its corresponding recursion stack, for Rescue, it is sufficient to keep track only of an e-NFA states to guide the ReDoS search.

Rescue is based on a three-phase gray-box analytical technique which finds for a given regex a timeout-triggering input string:

- In the *seeding phase*, given a regex (and corresponding e-NFA) a genetic (seeding) algorithm searches for a diverse set of strings that cover as many e-NFA states as possible regardless of the search time. For each string, a *effective prefix* and redundant suffix is kept to guide the cross-over operation and mutation in the *incubating phase*. The effective prefix is a prefix of the string s with a maximal number of characters in s that have been matched and the redundant suffix is the rest of the string s.
- In the *incubating phase*, a genetic algorithm searches for candidates with maximal ratio between the matching steps and their length. The new candidates are created applying mutation and crossing over operations to the seed strings from the seeding phase.
- In the *pumping phase*, the best candidate with the highest cost-effective ratio. First, all ReDoS-irrelevant characters are removed from the selected string. Then, the algorithm searches for a substring such that its pumping increase the number of matching steps the most. Finally, the best substring is pumped as many as possible to create a string of the given length l.

Example 4.3. The default setting of Rescue is that the maximum length of output string $l = 128$. So given a regular expression $(a|aa)^*\$$, Rescue returns a input string $a^{127}|$.

5 GENERATING REDOS ATTACK

In this section, we explain the algorithm for generating an eval text. The algorithm searches for optimal runs π_i in a search space of the underlying automaton A. The optimal run is the shortest run π such that:

- starts in the initial state q_0;
- explores as many unvisited states as possible;
- finishes in a state such that:
  - is rejected by A, and
  - all its successors are either final states, or visited states.

To gain the intuition how the text is generated, let's have a deterministic automaton A. The output will be lines of text such that each line corresponds to the labels of the run π_i. We start in the initial state of A q_0 since most of the matchers usually use optimization which allow them to skip a no-matching part of the input text.

We iterate over the transition leading from the initial state q_0 and choose the best successors according to the given criteria. We skip states that are not final. The reason behind this is that most of the backtracking tools try to find the match and if they do not succeed they conduct a backtracking search to find whether there exists any other path that matches the given input string.

The next requirement for the successor it that it is not already visited since our goal is to explore the whole state space of the automaton A as possible. Computing a DFA-state successor over a symbol is expensive, linear to the size of the NFA. Therefore, the modern matchers use caching of already visited parts of the DFA. If we will keep exploring new states we can slow them down.

In each iteration, using a transition we generate a new state. The minterm on the transition is used for generating the output character. We use a function *ChooseSymbol* which choose a random member at random from the set.

We keep exploring the state space till we found a state such all its successors are either final states or already visited states. In this case, we add a new line to the output string and start generating a new line of code.

The starting state of the new iteration is a state from the unvisited (already discovered) states. The unvisited state are on the edge of the part of the state space that has been not visited yet. So to

Algorithm 1: DFA-based space search

Input: A DFA $A = (\mathbb{I}, Q, q_0, F, \Delta)$.
Output: an ouptut string *string*.

```
1  unvisited ← {q0};
2  visited ← ∅;
3  str ← ϵ;
4  successors := {(q0 ↦ q0)};
5  while unvisited ≠ ∅ do
6      q ← ChooseBest(unvisited);
7      str ← str · Prefix(q, successors);
8      while not(timeout) do
9          T ← {(q, α, r) ∈ Δ|r ∉ F};
10         if T == ∅ then break;
11         else
12             (q′, α′, r′) ← SelectBest(T);
13             remove (q′, α′, r′) from T;
14             unvisited ← unvisited ∪ {r|(q, α, r) ∈ T};
15             visited ← visited ∪ {r′};
16             successors ← successors ∪ {(q ↦ r′)};
17             q ← r′;
18             str ← str · ChooseSymbol(α);
19         str ← str · newline;
```

explore the state space it is useful to use them to continue in discovering new states. Moreover, we keep the successors of all the states so that we can easily find a prefix which leads from the initial state q_0 to the state q and hence we speed up the process of generating the output text.

The process finishes once we explore the whole state space, or when the timeout expires.

6 USING COUNTING-SET AUTOMATA FOR GENERATING THE EVAL TEXT

In the previous section, we introduce a general algorithm for generating an evil text which has potential to cause ReDoS attack for most of the matchers. However, to create more efficient method we use counting-set automata. The key feature of the automata is enconding the

7 EXPERIMENTAL EVALUATION

We have implemented our approach in a C# prototype called `Cavil` available at [**?**] and evaluated its capability of generating text causing efficiency problem (ReDoS attack) for the state-of-the-art regex matchers on patterns that use the counting operator. We use the following matchers': Google's `RE2` library [**?**][2], the standard GNU `grep` program [**?**] (version 3.3), the `.NET` standard library regex matcher from `System.Text.RegularExpressions` [**?**], and Symbolic Regex Matcher (`SRM`) [**?**].

Let us shortly summarize how the tools work. The main algorithms of `RE2` and `grep` implement optimized versions of the Thompson's on-the-fly determinization where the constructed DFA states are cached. The construction has a bound on the size of the DFA—if the bound is reached, the so-far constructed DFA states are flushed to avoid consuming too much memory. In some situations when caching is found ineffectual, `RE2` turns the caching off, and the performance can drop even lower (see the description in [**?**] for details). We note that `RE2` rejects an input regex if it

[2]We used the version 2019-01-01 of `RE2` via the command line interface `re2g` from https://github.com/akamai/re2g.

contains a counting operator with a bound bigger than 1,000. SRM is based on *symbolic derivatives* constructed on the fly, also in the spirit of the Thompson's algorithm, and, likewise, bases its efficiency on caching (in fact, SRM is quite close to an implementation of the Thompson's algorithm over CAs with caching). The .NET matcher uses a backtracking algorithm over NFAs, while our Cavil eagerly constructs a deterministic CsA for the input regex. The former four are mature tools, and especially RE2 and grep contain many high- and low-level optimizations, such as using the Boyer-Moore algorithm [?] to skip over many characters that are known to not be a part of a match. RE2 and grep are compiled programs while Cavil, SRM, and .NET run within the .NET Framework (therefore, they have some inherent overhead due to the *just-in-time* compilation at start-up and its inability to use advanced code optimizations, as well as garbage collection). Note that even though the tools based on the on-the-fly subset construction (RE2, grep, and SRM) are linear to the length of the text, they still take space exponential to the counter bounds in the worst case, by creating sets of the size linear to the counter bounds, exponential to their decadic encoding used in the regex.

We compare our tool against the-state-of-the-art generators that are mainly focused on backtracking matchers, namely rxxr2 [?], regexStatic [?], and rexploiter [?]. These generators use different algorithms to generating the eval text.

The detectors may consume excessive time while analysing the regex and generating the eval text, hence, we limited the generators to 10 minutes. Notice, that all of these tools are research prototypes, so they do not support all regex featerues.

The detectors generate the output text in a form of *prefix*, *pump* and *sufix*. To test the accuracy of each generator's predicted eval text, we run the state-of-the-art matchers on the generated text. First, to determine the number of pumps we should to create the input text from the generated output of the generators we measure how long each regex take to match a sequence of malign inputs with varying numbers of pumps. The number of pumps ranges from 1, 10, 100, ..., 100.000 pumps. At the end, we decided to set the number of pumps to 100.000 pumps. These number of pumps was sufficient to cause that the regex match took more than 10 seconds on a match.

We run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz running Debian GNU/Linux (we use the Mono platform [?] to run .NET tools). To avoid issues with generating exact matches, which might differ for different tools, the tools were run in the setting where they counted the number of lines matching[3] the given regex (e.g. the -c flag of grep).

7.1 ReDoS Resiliency

Our main experiment focuses on generation text which causes a ReDoS attack. The regexes used for this experiment were selected (1) from the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [?]; (2) from databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, Snort [?], Bro [?], Sagan [?], and the academic papers [??]; (4) the RegExLib database of regexes [?] which is a website dedicated to regexes for various DSLs; (5) regexes from posts on stackoverflow [?]; and (6) industrial regexes from [?], used for security purposes. From these, we created our set of benchmarks by the following steps:

(1) From all the regexes, there were 88,000 regexes containing a counting operator. We filter 2,080 regexes with sub-expressions similar to a{0,1} or even just a{1} (there are surprisingly many of these occurring in practice where the use of a counting loop is unnecessary).

[3]We consider the standard semantics of "matching" used by grep, i.e., a line matches a regex R if it contains a string that is in $\mathcal{L}(R)$, unless it contains start-of-line (^) or end-of-line ($) anchors, in which case the matched string needs to occur at the start and/or at the end of the line respectively.

(2) Then, we filtered 7,980 regexes that contain nested counting (*We do not want to bother with them.*) and 7,100 regestes that are not supported by our tool (containing lookarounds (? =), (?!), (? <!), backreferences \1, non-capturing groups (? :), lazy quantifiers ??, ∗?, ?, , +?, possessive quantifiers ?+, ∗+, ++, ∗), and other unsupported symbols, such as \b, \h, \g, \p, \B, \G, \P, etc..
(3) We selected regexes that contained counting loops whose sum of upper bounds was larger than 20. This let us focus on regexes where the use of counting makes sense. Moreover, we also removed 17 regexes with counters bigger than 2,147,483,647, which cannot be handled by our tool. This left us with *5,300* regexes.

Our generator `Cavil` generates an evil text in a form of one to several lines of text. The generating phase finished after timeout expered or the whole state space is explored. Then, we make multitimes copies of the text to create ∼10 MiB long input texts for the matchers. From all benchmarks, there were *34* regexes where the generator hit the timeout of 10 minutes. This was caused not by the counters but rather by many "`|`" and "`?`" operators.

7.2 Running Matchers

We ran all tools on the generated benchmarks (counting the number of lines of the input text matching the regex). The timeout was set to 5 s. If the matcher needs more than 5 seconds to match the input text we consider the input text to be a ReDoS attack. We merged the results according to the techniques that the matcher used (backtracking, . . .).